JOVE: Java as it Happens

Steven P. Reiss and Manos Renieris
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
{spr,er}@cs.brown.edu

Abstract

Dynamic software visualization is designed to provide programmers with insights as to what the program is doing. Most current dynamic visualizations either use program traces to show information about prior runs, slow the program down substantially, show only minimal information, or force the programmer to indicate when to turn visualizations on or off. We have developed a dynamic Java visualizer that provides a statement-level view of a Java program in action with low enough overhead so that it can be used almost all the time by programmers to understand what their program is doing while it is doing it.

CR Categories: D.2.6 Graphical environments, D.2.5 Debugging aids.

Keywords: Dynamic software visualization, run-time monitoring, instrumentation.

1 Introduction

We want to be able to understand the behavior of our software. In particular, we want to be able to understand what the software is doing when performance issues arise, when it undergoes unexpected behavior, and when it interacts with the user or the outside world in a particular way.

Understanding such behavior is difficult at best. Software is large, consisting of tens of thousands of lines of code all of which can interact in arbitrary ways. Software involves high-speed execution. Code executes so fast that it is virtually impossible to look at the fine grain behavior of a large system in any meaningful way, especially as it is occurring. Today's systems are long running. The performance and other issues that arise in a modern server system are temporal, arising only occasionally, and are typically dominated by the sum total of the other behaviors of the system. The performance of a particular event or interaction is difficult to isolate. Finally, today's system are complex. They have to deal with multiple threads of control interacting in non-obvious ways and take for granted such complex entities as garbage collectors and library functions as XML parsing.

We feel that the best way of understanding such software systems is to be able to look at a detailed synopsis of their behavior as it happens in such a way that the types of things that we might be looking for, in particular performance issues, thread interactions, and unusual behavior, stand out. Doing this as the software executes lets us correlate what is going on in the software with the appropriate external events (user interactions or other programs) that trigger the corresponding behavior. Using visualization provides a high-bandwidth channel from the data to the observer, letting us use our visual abilities to quickly find unusual patterns and letting the tool use appropriate visual cures to highlight potential items of interest. Using appropriate synopses compresses the data into something meaningful while letting us isolate what might be of interest.

To this end, we have created two visualization systems aimed at understanding the behavior of complex Java programs. Our earlier system, JIVE, looked both at class-level behavior and at the interaction of threads [16,18]. It demonstrated that it was possible to do dynamic visualization of Java with very low overhead (a factor of 2) while producing meaningful and useful views of the software. JIVE summarized information in terms of intervals of 10 milliseconds or more (under user control). For each class or collection of classes where appropriate it collected the number of calls of methods of the class, the number of allocations done by the class, the number of allocations of objects of the class, and the number of synchronizations done on objects of the class. In addition, it tracked the state (running, running synchronized, waiting, blocking, sleeping, doing I/O, or dead) of each thread, the amount of time spent in each state, and any synchronizations between threads. This information was displayed dynamically in a compact form that highlighted classes and threads that had unusual behaviors. The system also kept track of the history of the run so the user could revisit or replay the history to further examine the behavior.

The key aspects to making JIVE successful were to:

- Minimize the overhead so the system could be used on any program at any time. This includes avoiding any new synchronizations and thus affecting program behavior.
- Maximize the information gathered and displayed so that complex, interacting patterns could be identified and so it was more likely that the behavior to be understood was represented in the display.

- Provide history information so the user can replay the execution or revisit interesting execution points.
- Key the display so that the types of behavior that are likely to be of interest are highlighted using appropriate visual cues such as color and size.
- Let users adapt the display cues to their particular problems.

While JIVE is very useful for a high-level understanding, it does not provide enough detailed information to address specific problems such as where execution is occurring in the code, why a particular thread is using all the execution time. or even what each thread is actually doing. In particular, we needed:

- Information on where in the source execution is actually occurring so we can determine where the application is spending its time and why.
- Information that relates instruction execution to particular threads of control so we can identify what each thread is doing and not just what state it is in.

This led us to develop an alternative system, JOVE, that gathers data over intervals in terms of basic blocks on a per-thread basis, and then provides a corresponding dynamic display that shows what is going on in the program as it happens. JOVE meets the requirements that made JIVE successful in that it has small overhead (a slowdown of 3-4), lots of available information, a configurable display that highlights unusual information, and a history mechanism to let the user navigate in time over the run.

In this paper we describe the JOVE system. We start with an overview of the system and an example. Then we briefly describe related work. Next we explain how we gather the necessary data with low overhead. Then we detail the display we have developed for this data and explain how the user can configure the display to highlight different behavioral aspects. Finally we outline future work.

2 An Overview of JOVE

JOVE, like JIVE, works by gathering data from a running Java program and displaying a summary visualization of that data that shows what the program is doing as it occurs. JOVE splits the subject program's execution time into intervals of approximately 10-20 milliseconds. Within each interval, Jove records what the program is doing in each thread. While there are several ways of doing this, the standard for counting and performance analysis is to use basic blocks [8,20]. A *basic block* is a segment of straight-line code with no internal branches. This means that if the block stars executing, then (in almost all cases) all the instructions in the block will be executed. One can get counts of the number of times an individual instruction is executed by simply determining the number of times the block it is contained in is executed. From this information and compiler information as to what instructions in the executable correspond to what lines in the source, we can use basic block counts to determine the number of times a particular line is executed and the number of machine instructions or byte codes that are executed for each particular line.

In particular, the atomic datum that JOVE records is a tuple <*I,T,B,c*> where *I* identifies the interval, *T* identifies the thread, *B* identifies a basic block, and *c* is the number of times block *B* was executed by thread *T* in interval *I*. JOVE extends the basic block counts to counts of allocations and executed instructions from a static analysis of the basic blocks. It then accumulates all these statistics in multiple ways, for example by file and by thread, and maps the results to visual elements of an information-dense display.

JOVE's main visualization window consists of several vertical regions, each corresponding to a file of the subject system. Each region is divided into two subregions, a portion on top for thread information, and a portion on the bottom for block information. In the bottom region, there is a horizontal rectangle for each basic block. These rectangles are ordered by the number of the corresponding source code line(s). These two elements of the visualization, the vertical regions and the horizontal lines of "source code" resemble SeeSoft [3]. Unlike SeeSoft, however, most elements of the display change dynamically according to the data collected in the last completed interval. The result is effectively a movie of the program in action.

While much of the display shows what is happening in the last interval the overall file display reflects the totals for this file up to the current interval. The width of each vertical region corresponds to the percentage of the number of instructions executed in the file over the entire run. We found that associating the width of these regions with the percentage of time spent in only the last interval resulted in abrupt display changes. Similarly, each file region is colored with a darker and lighter color whose boundary reflects the number of allocations performed by the code in this file over the whole run.

The top, thread subregion of each file display contains a pie-chart. The whole of the pie-chart corresponds to the total time spent in this file during the current interval. The pie-chart is split into sectors with each sector corresponding to the portion of the time spent within a particular thread. Colors are used here to differentiate the threads. When viewed during the run, the pie chart shows how different threads are executing in the different files over time.

The basic background color of each vertical region encodes three different statistics. The hue is associated with the number of instructions executed in the current interval using a green to yellow color model; the saturation is associated with the number of threads executing in that file during the interval; and the brightness reflects the number of allocations done by blocks in that file during the interval. If no code in the file was executed during the interval, the color defaults to gray. Our previous experience has shown that while saturation and brightness are not ideal for such displays, they can effectively highlight interesting cases. An alternative, using patterns, is currently too slow to run in real time on most machines.

Each of the basic blocks of a file is depicted by a horizontal rectangle within the lower portion of the file region. The height of the rectangle corresponds to the number of actual source lines

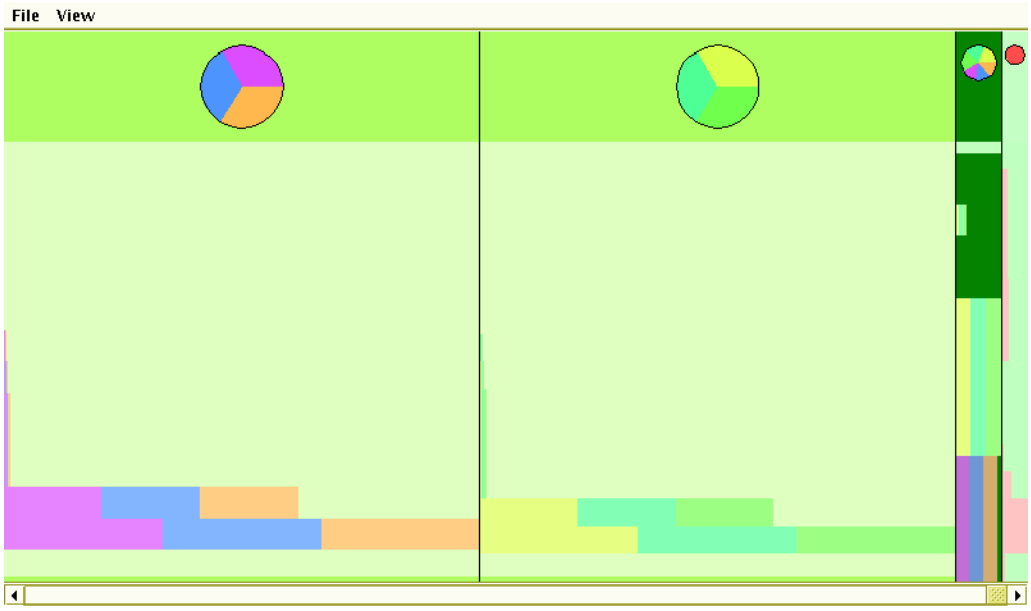


FIGURE 1. Balanced Producer-Consumer program run with JOVE.

(resulting in a correspond of pixel rows to lines of code). The width of the rectangle corresponds to the number of the instructions executed in the basic block during the last interval. The line is split in multiple colored regions, each corresponding to a thread, and color coded to agree with the pie-chart.

The JOVE display attempts to provide at a glance a wide range of information keyed to the current program execution. The different display techniques were chosen so that potential problems, such as thread contention and performance bottlenecks, stand out. It provides the viewer with a quick overview that facilitates identifying problems when they occur; and lets the user zero in on those problems by looking at the visualization more closely.

3 An Example

To understand how JOVE works, consider a simple Producer-Consumer program adapted from [1]. This program consists of a main class, two classes inheriting from *Thread*, *Producer* and *Consumer*, and a *CubbyHole* class to hold the intermediate data. Each producer inserts 100 integers in the *CubbyHole*; each consumer extracts 100. In addition, producers and consumers execute a small idle loop after each integer insertion or extraction. The cubbyhole methods are synchronized, meaning that only one insertion or extraction can be in progress at any given time. In our version the cubbyhole is built around a vector, which means that insertion of integers is always possible, but naturally, extraction is only possible from a non-empty cubbyhole. The main program starts an equal number of producers and consumers.

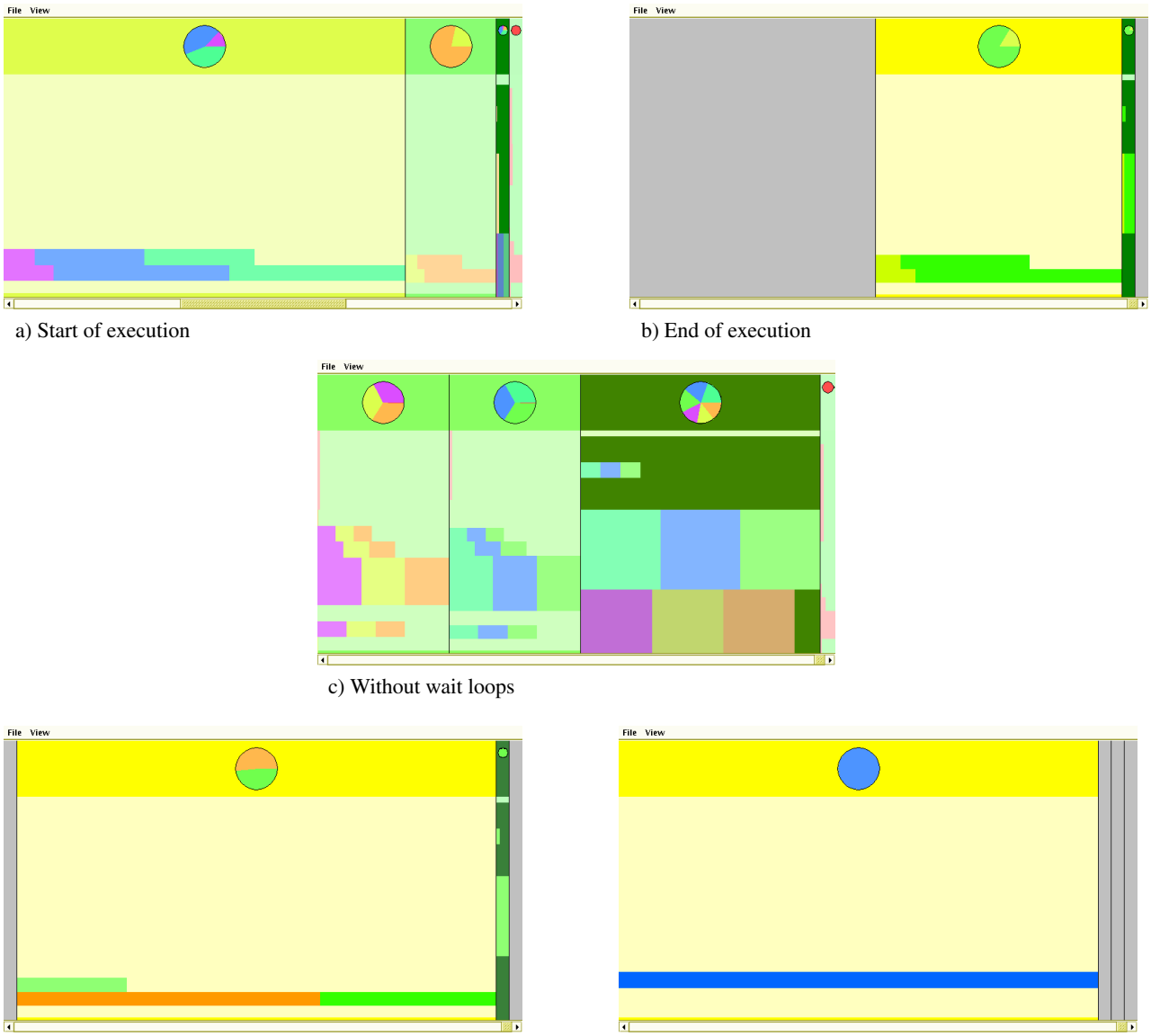
We first run the program (under JOVE) with three producers, three consumers, and a small delay value. The program then runs in a single JOVE time interval, and JOVE only produces one picture, Figure 1. The program executes in eight threads: one for each producer/consumer, one for the cubbyhole, and one for the main, initial thread. Since there are four files in the subject system, the JOVE window is split in four panels. We can immediately say, by their width, that the first two panels consume most of the execution time, and that they execute for about the same time.

The first panel corresponds to the Producer file. As shown in the pie-chart, the producer spends its time equally in three threads, the “blue”, “purple” and “orange” threads. Most of the time is spent in the idle loop of line 17, again split equally between the three producer threads. The second panel corresponds to the Consumer file and shows identical behavior, except, of course, the code executes in different threads.

The third panel represents the CubbyHole, which executed code in all 8 threads (although so little time is spent in the main thread that it is hard to discern that fact in the pie chart). CubbyHole spends time primarily in three regions of the code: Line 11, which tests whether there is an item available; Lines 17-21, the actual removal of an item, and lines 26-29, which the insertion of an item. As evidenced by the thread color correspond, the first two blocks are executed on behalf of the consumer code, while the last block on behalf of the producer code. The saturation of the last block in CubbyHole is higher, because all the allocation in the system happens in that portion of the code.

Lastly, the fourth panel corresponds to the main thread and the driver code. It executes for about 10% of the time in the “red” thread and with some allocations (of the thread objects).

In the second example, we also used three producers and three consumers. In this case, however, the number of idle loop iterations was much higher. As a result, two transient effects become obvious. In Figure 2a towards the beginning of the execution, the producer threads take more time than the consumer threads



a) Start of execution

b) End of execution

c) Without wait loops

d) With a fast producer

e) With a fast consumer

FIGURE 2. Different JOVE views of the Producer-Consumer program.

because they can always add items to the CubbyHole, while the consumer threads need to wait until an item is available. This is reflected in the wider area allocated to the producer thread. In Figure 2b, towards the end of the execution, the width of the producer window is equal to the width of the consumer window, since, at the end of the execution, both kinds of threads have executed for about the same time, since the dominating idle loops are equal. However, during the latest interval, no producer thread executed, and therefore the producer region is blank.

Note that in all examples so far, the cubbyhole executed for a very small amount of time, because the idle loops dominate execution. When the idle loops' iteration count is set to zero, Jove produces Figure 2c.

As a last example, consider the situations where either the consumer or the producer is much slower than their counterpart. In Figure 2d the producer is faster, and as a result, the first region is much smaller than the second. The situation is reversed in figure Figure 2e.

Another example of the use of JOVE can be seen in Figure 3. Here we are visualizing a pinball program as it runs. The program has one thread doing the physics computations 1000 times a second, one doing 3D graphics display, one doing sound, and one handling keyboard input.

From Figure 3 and tooltips denoting what we are looking at, we can detect several things about the program and its execution at this point in time. First, from the relative widths of the different file blocks, we can see that most of the execution time is spent in

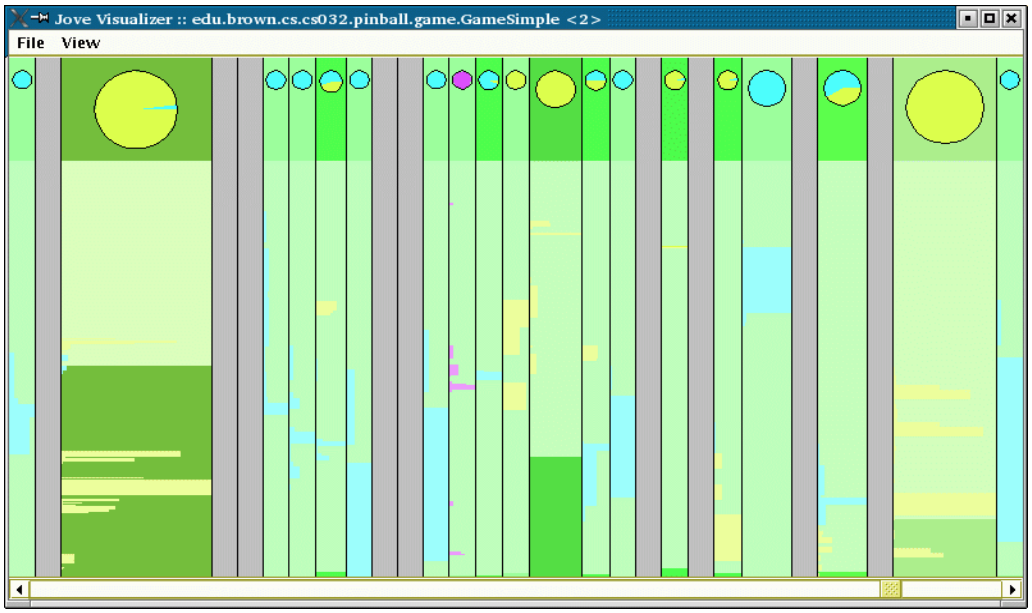


FIGURE 3. JOVE visualization of a pinball program.

the large block on the left (ComponentBase.java) which represents a generic object on the pinball board. Within this file, most of the time is spend in the area designated by the yellow lines at the bottom, notably within the *intersectLline* method. Most of this execution time derives from the computation thread (yellow). Note that a small fraction of the execution of this file is in light blue, representing the drawing thread. The particular methods here set the color and material based on the component.

There are several files where execution is split roughly evenly between the drawing and computation threads. Most notably, the file panel toward the right. This file implements wall components. The blue here represents the routines for drawing the wall; the yellow represents the routines for computing intersections. We can tell from a glance that there is no overlap. Moreover, we can also tell from the small highlighting of this panel, that the code here does few allocations.

Finally, while most of the execution is dominated by the computation and drawing thread, there is one small file containing purple. This represents the sound thread and is an indication that sound uses few computational resources in the program.

4 Prior Work

There have been a large number of different systems that provide visualizations of the dynamics of a program. Ours are different in that they attempt to provide high-level program-specific information in real time.

Two views of our previous dynamic visualizer JIVE are shown in Figure 4. The left side of each view shows the various classes and packages at the level of detail chosen by the user. Each class box contains a rectangle whose height indicates the number of calls of methods of that class, whose width represents the number of allocations done in those methods, whose hue represents the number of allocations of objects of that class, and whose saturation reflects the number of synchronizations. The right side of the top view contains a region for each thread. In the region is a stack of rectangles that show the states the thread is in over the interval, and what fraction of the state is represented by that thread. The right side of the second view displays the alternate thread view

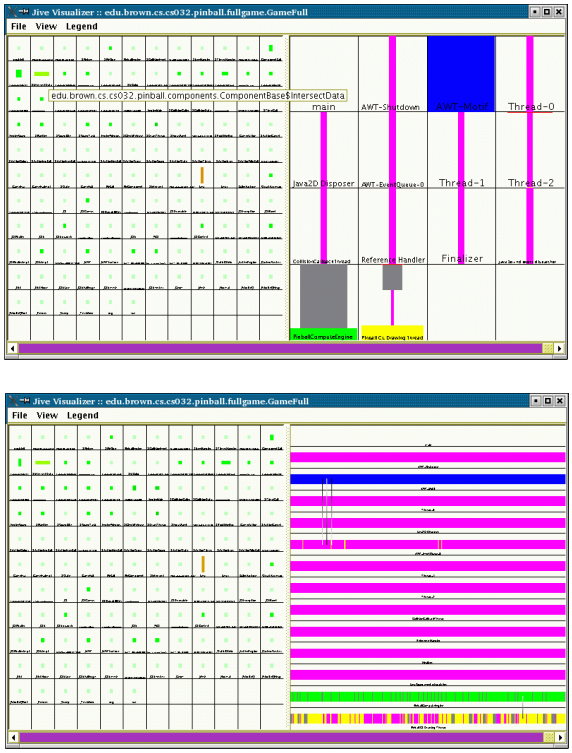


FIGURE 4. Our previous visualizer, JIVE.

where the state of the different threads is shown on a time axis with vertical bars indicating synchronizations among the threads. In both cases, the scroll bar at the bottom is colored to show the (dynamically computed) phase of the program.

Outside of our work, perhaps the most prominent effort is IBM's Jinsight [9-11]. Jinsight typically runs by collecting detailed trace data as the program executes and then, after execution is complete, letting the programmer understand execution at a very detailed level using a variety of views based on the trace. Trace collection, however, is not that efficient, requires a suitably modified JVM (and the program to work with that particular JVM), and is typically not the type of thing one would use all the time. Recent work on Jinsight has been aimed at letting the programmer identify just those portions of the program for which tracing should be done. This provides for almost immediate visualizations, but assumes that the programmer knows what to look for in advance.

The program visualization group at Georgia Tech has implemented several visualizations that provide insights into program execution using program traces [4,6]. Similar systems include PV from IBM [5], and the dynamic aspects of the Bloom and Almost systems [14,15,19]. The problem with these trace-based analyses is that they require the programmer to take the extra effort to run the system with tracing and often are both difficult to use and run too slowly to be practical. Our goal was to get as much of the information that these tools provide as possible without the considerable overhead that they incur.

Another set of relevant tools are performance visualizers that provide insight into what the machine is doing while the program is being run. These range from standard operating-system based performance tools such as those incorporated in Sun's workbench toolkit or IBM's PV system, to viewers that concentrate on some specific aspect of execution. In the later category, one finds dynamic visualizations of thread behavior [2], visualizations of heap, performance and input/output in the FIELD environment [12,13], and the large number of different visualization of the behavior of processors and messages in parallel systems culminating in the various MPI visualization tools such as *upshot* or *xmpi*.

There are also commercial tools that provide some dynamic performance information as the program is running. For example, Borland's OptimizeIt, provides graphs of cpu time, memory behavior, loaded classes, and garbage collection behavior. It also lets the programmer create snapshots and then investigate both the object graph for memory problems and a dynamic call graph for performance information.

Finally, we note that dynamic visualization is nothing really new. Back in the 1960s programmers worked on understanding what their program were doing either by looking at the console lights or the performance meter of the system (on a GE635) or by placing a radio next to the system and listening to the different types of static that were generated.

5 Implementation Overview

The overall operation of JOVE is based on our previous experience with JIVE. The system consists of four basic components, control, setup, information gathering, and visualization.

The control component presents a basic interface to the user. It lets the user define the system that is to be run, provide or change arguments to either the Java interpreter or the user code, identify which classes or packages should be monitored and which should be ignored, and change the settings of how statistical properties are mapped to graphical properties of the visualization.

The setup component is an independent process that takes a set of Java classes and a class path, identifies all the classes that should be monitored, and then patches those class files by inserting appropriate code to do the monitoring. Our current patcher makes use of IBM's JikesBT byte code toolkit [7]. This component produces two outputs. The first is a jar file that contains the modified class files. This is used to replace the original class files when the application is actually run. The second output is a descriptive file that itemizes information about each basic block in the program, including the containing method and class, the source lines, the number of instructions in the block, the number of allocations in the block, and the types of objects allocated by each of these allocations. This second file is used by the visualizer to translate raw counts from the basic blocks into more meaningful information for the user.

The information gathering component is a small library that is loaded with the user's program. This library is called initially and by the instrumented code. It is responsible for keeping track of counts and providing the appropriate data dynamically to the visualizer. Details of the setup and information gathering components are provided in the next section.

Finally, the visualization component provides two displays. One is a transcript of the program's input and output. The second is the main display of the collected information as previously described. This component is in charge of storing and accessing the dynamic information, of creating an appropriate display based on the user's preferences, of dynamically updating that display as the program runs, and of providing time-based access to the data. This is described in the subsequent sections.

6 Getting Detailed Trace Data

The key to a successful real-time dynamic visualization system is obtaining the appropriate trace data with minimal overhead. For JOVE this means getting counts of how many times each basic block is executed for each thread in some interval.

In our previous work we determined that the most efficient and the only practical way of getting trace data with small overhead from Java was to patch the java byte codes before execution to add the appropriate information gathering instructions. In JIVE we patched every function entry and exit for the classes we were monitoring with a call to a function that incremented an appropriate counter inside the current buffer. For functions that repre-

sented a thread state change, we also recorded the current thread and time and new state when the function was entered and restored the prior state when the function exited. For library classes we created stubs for each external entry to the library and had that routine do the counting so we would not have to patch the library code itself.

While very efficient, this did not provide the detailed information that we wanted to gather and display in JOVE. In particular, it only gathered information at the method level, not the basic block level, and, more importantly, it did not associate any of the counts with the appropriate thread.

In some ways, gathering information at the basic block level is easier than at the method level. In particular, in JIVE not only did we have to patch every method entry and exit, we also had to worry about allocations and abnormal exits through exceptions. Neither of these needs to be done if one is doing basic block tracing. First, information about allocations is encoded in the basic block information since we can statically determine in almost all cases what objects are going to be allocated when a block is executed. Second, since we statically know the method location of each block and whether that block represents an exception handler, we can detect exceptions from the output sequence.

The first problem we had to tackle in getting basic block information for JOVE was to have the counts be associated with the current thread. Our previous work showed that determining the current thread at each method entry was too expensive. Our solution here was to add an additional parameter to each routine containing the current context which includes the current thread and the count information for that thread.

Adding a parameter creates some problems. First, any routines that implement standard interfaces such as callbacks or abstract methods of classes that aren't patched, cannot be changed. Second, all calls to any modified method need to be changed as well. Where these calls occur in modified code, this can be easily done; where they occur in unmodified code, this is not possible.

Our solution was to create a shadow routine for each routine that was being traced. We changed the name of the original routine, added the context parameter, and then added code to the routine to call an appropriate method on each basic block entry. We then created a new stub routine with the original name and parameters. The new stub routine computed the context (by finding the current thread and then looking up the context based on this) and then called the modified original routine. We then modified all calls to routines from the patched code to be calls to the corresponding renamed routine.

This solution meant that almost all calls that were made in the modified code used the modified original routines and did not have to compute a new context. Moreover, we are able to handle correctly calls from unmodified code and routines that implemented interfaces or abstract methods.

The next problem was to do efficient counting and reporting of blocks for each thread. Here we had the context for each thread contain preallocated count buffers sized to hold counts for each of possible basic block in the monitored source. The code on basic block entry just increments the corresponding counter for the current context. Each context triple buffers these counts. One buffer that holds the current counts. A second buffer is used to hold the data that is currently being processed for sending to the visualization front end. A third buffer, representing the previous interval, is available for comparison and deltaing from the second when starting up (to track threads whose state did not change during the interval, for example), and then has all its counters cleared so that it is available at the next interval.

The actual collection and reporting of the data to the visualization front end is done by a separate thread that is run off a timer that computes the intervals. This thread handles the buffer swapping, produces an XML message that summarized all non-zero counts, and sends this message over a socket to the visualizer.

7 Detailed Dynamic Visualization

Information gathering yields basic block counts for each interval for each thread or context. For display purposes, we accumulate this information by context and globally for each source file, accumulating all blocks for a given file and mapping the basic blocks to the corresponding file lines.

The next issue that arises is how to effectively display this information. We first noted that there are three basic types of information that we can display. First, we have information about the individual basic blocks (and hence source lines). Here we have the number of times they are executed, the number of allocations done, and the number of byte codes that were executed, both for each individual thread and for all threads together. Second, we have information about files. Here we have total counts of these various statistics from all the blocks in the file again associated with different threads. Third, we have the various statistics accumulated by thread.

We wanted to display all three types of information on the same display. In doing so, we wanted to ensure that we could display as much of the data as possible, and that interesting values, notably possible performance problems, thread conflicts, and interesting changes in behavior, would stand out.

As noted in Section 2, the is organized into vertical file regions, each of which contains a pie-chart display for contexts and a SeeSoft-like display of basic blocks. Global file information is reflected in the file region size, background color, and the division between the dark and light background. The role of contexts in the file during the interval is the pie chart. Rectangles in the basic block region reflect context, allocations, and the number of instructions executed during the interval.

Central to the JOVE display is the ability to show multiple dimensions of data in a meaningful way. The general file display shows five different statistics using width, hue, saturation, brightness, and the position of the separating line. The pie chart shows the various contexts by color and can reflect different statistics about that context (e.g. instructions or allocations) in the pie chart

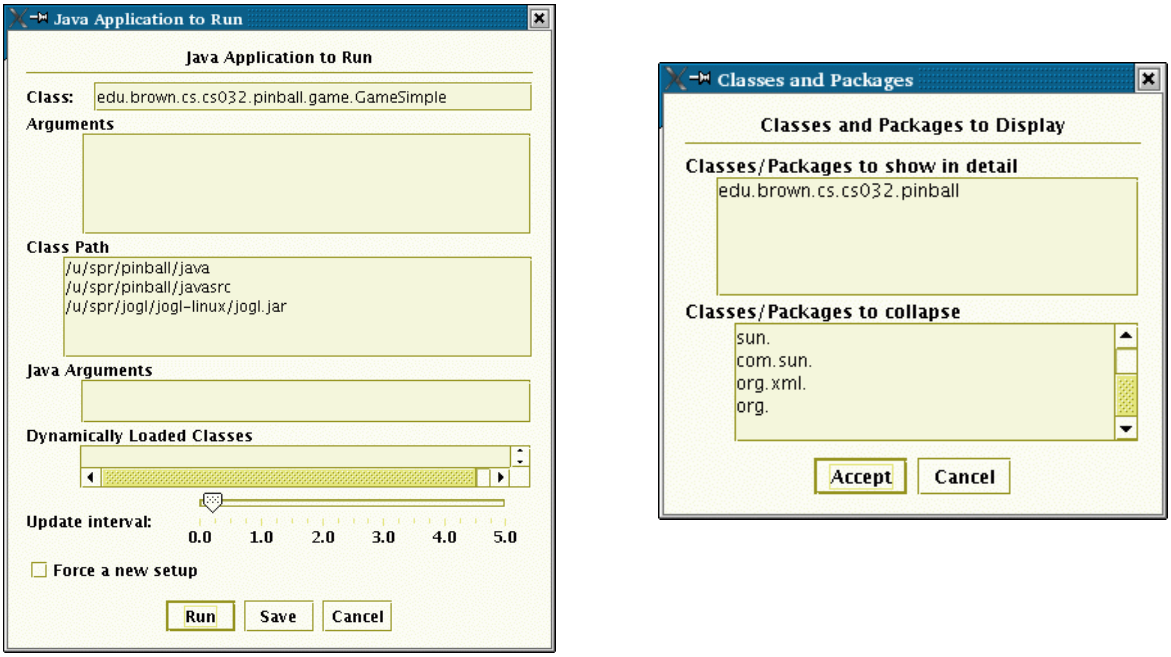


FIGURE 6. Dialog boxes for controlling the execution and specifying detailed and library classes.

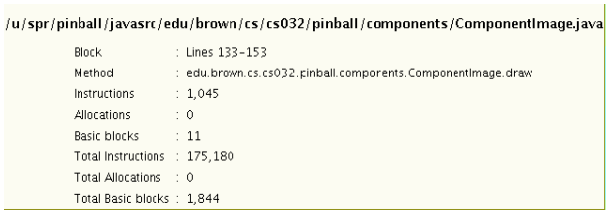


FIGURE 5. Tooltip showing block details.

size. The basic blocks can encode up to five statistics in their width, height, hue, saturation, and brightness.

While the display provides a lot of information, the programmer needs to correlate it back to the source program. Rather than attempting to label everything in place (where the text would often be too small to read), we use tooltips to provide detailed information to the user. Figure 5 shows the tooltip that results from placing the mouse over the large blue area in Figure 3.

8 Customization

JOVE is designed so that the user use can adjust the program to obtain the most appropriate display for the application at hand. We normally run JOVE from the command line, where we have installed a *jove* command that basically replaces the normal *java* command, but runs JOVE on the application. Within JOVE, the user can use the dialog boxes shown in Figure 6 to set the various command line arguments, to change the interval size, to specify classes that are dynamically loaded and not statically detectable but that should be monitored, and to identify which packages and classes should be monitored and which can be ignored.

The user can also control what statistics are displayed in what way using the display properties dialog box shown in Figure 7. The user can choose total or immediate values for each of the statistical properties. Moreover, the user can choose either linear, square root, or log scaling to take into account different distributions of values for those properties. Other user-settable items include the color models (red-violet, yellow-red, green-red, yellow-green, blue-red, or green-yellow), whether the statistical values should be inverted or not, whether blocks should be expanded to take up the full range of the file box or just size proportional to the file size, and the way that shading should be done for file displays.

9 Data Structures

One of the key implementation problems in JOVE was storing the data coming from the application in such a way that we can rapidly compute the display at each interval. On a long running program, there can be a large number or intervals, each of which has significant data. For example, in the relatively small pinball example, there are about 3000 basic blocks that have to be tracked, maintaining two values (local and total) for each interval for each of the fourteen active threads. Moreover, we needed to provide access to summation counters for files and threads. This is more complex, as we need to accumulate the different statistics (allocations, instructions, and block counts) separately since they can't be directly derived from the raw counts as they can for basic blocks.

To store this data we create a vector of intervals. Each interval is represented as two hash tables, one for the local data for this interval and one for the total data up to and including this interval. Each hash table is indexed by the block or file where the

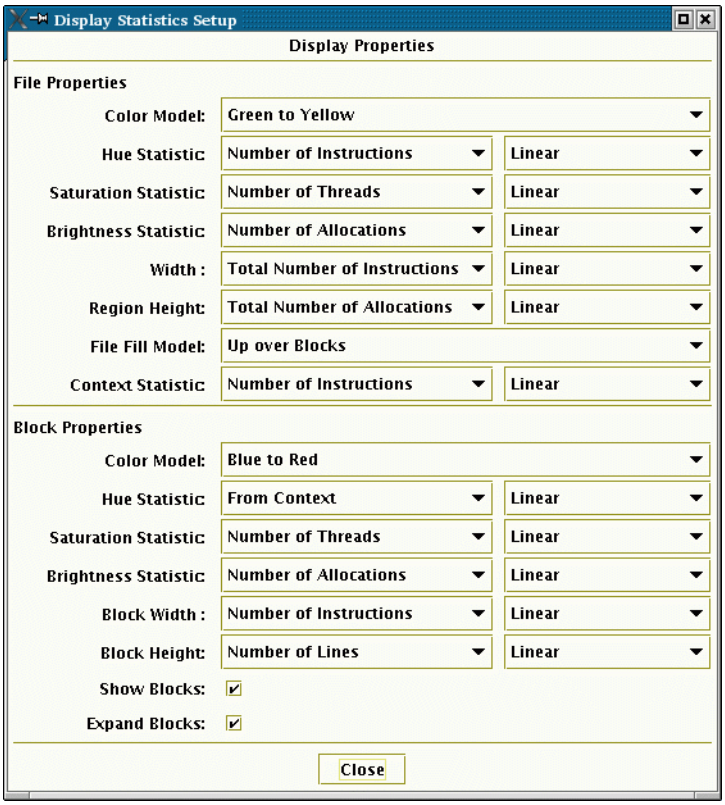


FIGURE 7. Dialog box for controlling display properties.

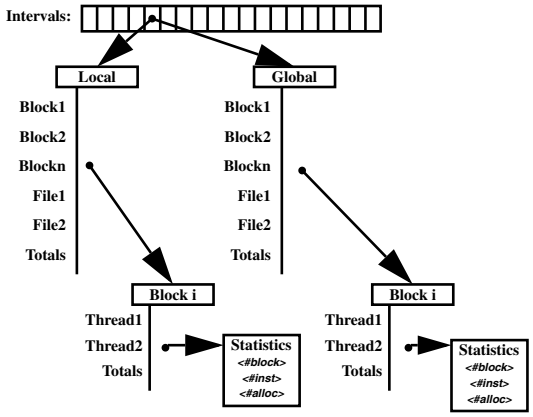


FIGURE 8. The data structure used to store count information.

associated value is another hash table. For each individual item, this second hash table maps the context (thread) to a block of statistical values representing the different counts. A special context value is used to represent the global context. This is illustrated in Figure 8.

The raw data that is received from the information gatherer for each interval is at the block level, providing each non-zero count of a block for a particular thread. We first use this data to set up the local tables. Each non-zero count generates a statistics entry for the corresponding block and thread (going through the two levels of hash table). The total for all threads is computed as a separate statistic from the local totals; similarly, each block count is also accumulated into the corresponding entry for its file indexed in the first hash table. Finally, the total of all entries is also computed. The result is a complete set of the non-zero statistics for each interval that can be rapidly indexed by block or file and thread or any combination thereof.

Totals are kept in a similar fashion, but are computed in a different way. We generate the totals after we have computed the local values for the interval. For each entry in the previous totals that has no corresponding local entry, we simply link to the previous statistics block. Otherwise, we create a new totals entry (again using two-level hash tables) that contains the sum of any previous total entry and the new local values.

The result is a fairly space-efficient storage mechanism that provides rapid access to all the available statistics and takes into account the fact that most of the counters will be zero. Using it, we are able to save about an hour's run of the pinball program in memory and then rapidly scroll over the display using the history bar to get an overview of the run.

10 Future Work

While not perfect, our efforts show that detailed dynamic visualization of real applications is possible and may be practical as a default way of running the application. The program runs with a slowdown of a factor typically between 3 and 4 depending on the

structure of the application. Given the wide performance range of today's machines, this seems to be quite acceptable.

There are several directions for future work. First, we note that JIVE and JOVE provide complementary information about program execution. While JOVE provides detailed information about particular source files, JIVE provides information about thread states, synchronization, and about the behavior of library classes. Ideally, one wants all the information from both tools available for a single run. This seems possible, with the major difficulty being avoiding slowing down the computation any more that it already is.

A second involves addressing properties of the execution that neither JIVE nor JOVE deals with. One of these is actual timings. Current methods of getting detailed execution times are too expensive to be useful. We will be looking into using sampling and similar techniques to get good approximations without significant overhead. Another property that we would like to address is memory utilization. While we can track allocations, tracking deallocations in a garbage collected environment is more difficult. Simple solutions like adding a finalizer method to each class turn out to be too costly in the current JVMs. Here we are looking into using the internal Java profiling tools (JVMPI and the Java Management Package) in addition to the class patching we currently do.

A third direction involves associating program behavior with events, either input actions or program generated events. In a web service, for example, for is interested in understanding what processing is being done to handle a particular input request. One complexity here is that the processing might take place in several threads over an extended period of time. We plan to integrate our previous work in this area into the dynamic visualizations of JOVE [17].

A fourth direction is to make it easier to correlate the display with text. Here we plan to allow the user to select a particular file that is of interest and provide a separate window that correlates the display for that file with the actual text of the file.

In conclusion, we have found JOVE is capable of providing dynamic information about real Java programs as they execute. Its abstract, multidimensional display provides lots of information and graphically highlights potential problems. It offers a new tool to aid in the dynamic understanding of software systems. JOVE is available from our website at http://www.cs.brown.edu/people/spr/research/bloom.html as part of the BLOOM package.

11 Acknowledgements

This work was done with support from the National Science Foundation through grants CCR021897 and ACI9982266.

12 References

1. Mary Campione and Kathy Walrath, *The Java Tutorial*: *Object-Oriented Programming for the Internet*, Addison-Wesley (1996).

2. Bryan M. Cantrill and Thomas W. Doeppner, Jr., "Threadmon: a tool for monitoring multithreaded program performance," *Proc. 30th Hawaii Intl. Conf. on Systems Sciences*, pp. 253-265 (January 1997).

3. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft - a tool for visualizing software," AT&T Bell Laboratories (1991).

4. Dean Jerding, John T. Stasko, and Thomas Ball, "Visualizing interactions in program executions," *Proc 19th Intl. Conf. on Software Engineering*, pp. 360-370 (May 1997).

5. Doug Kimelman, Bryan Rosenburg, and Tova Roth, "Visualization of dynamics in real world software systems," pp. 293-314 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

6. Eileen Kraemer, "Visualizing concurrent programs," pp. 237-256 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

7. Chris Laffra, Doug Lorch, Dave Streeter, Frank Tip, and John Field, "What is Jikes Bytecode Toolkit," *http://www.alphaworks.ibm.com/tech/jikesbt*, (March 2000).

8. James R. Larus, "Abstract execution: a technique for efficiently tracing programs," U. Wisc.-Madison Computer Sci. Dept. TR 912 (February 1990).

9. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object- oriented software execution," pp. 329-346 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

10. Wim De Pauw and Gary Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," in *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, (1999).

11. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proc. ICSE Workshop of Software Visualization*, (May 2001).

12. Steven P. Reiss, *FIELD*: *A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

13. Steven P. Reiss, "Visualization for software engineering -- programming environments," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc Brown, and Blaine Price,MIT Press (1997).

14. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).

15. Steven P. Reiss, "An overview of BLOOM," *PASTE '01*, (June 2001).

16. Steven P. Reiss, "Visualizing Java in action," *Proc. IEEE International Conference on Software Visualization*, pp. 123-132 (2003).

17. Steven P. Reiss, "Event-based performance analysis," *Proc 11th IEEE Intl Workshop on Program Comprehension*, pp. 74-81 (2003).

18. Steven P. Reiss, "JIVE: visualizing Java in action," *Proc. ICSE 2003*, pp. 820-821 (May 2003).

19. Manos Renieris and Steven P. Reiss, "ALMOST: exploring program traces," *Proc. 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, (October 1999).

20. MIPS Computer Systems, Inc., *RISCompiler Languages Programmer's Guide*. December 1988.